МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

Лабораторна робота -2

Виконав студент

Групи ДА-02

Коваленко Олександр

Київ

КПІ ім. І. Сікорського

2023

**Мета роботи:** Розглянути базові примітиви синхронізації та їх особливості, в

залежності від обраної мови програмування. Розглянути підходи до побудови

ПЗ з використанням паралелізму та ознайомитися з класичною задачею

паралелізму у вигляді пулу потоків.

**Завдання:**

1. Ознайомитися з визначенням поняття пул потоків (thread pool),

використовуючи даний методичний посібник, або ж сторонні джерела.

2. Реалізувати власний пул потоків з характеристиками, зазначеними в

обраному варіанті. Зі спільних характеристик: пул потоків повинен бути

написаним коректно відносно обраної мови програмування, повинен мати

можливість коректного завершення своєї роботи (моментально, з

покиданням всіх активних задач, так і з завершенням активних задач),

можливість тимчасової зупинки своєї роботи. Операції ініціалізації та

знищення пулу, додавання та вилучення задач в чергу повинні бути потіко-

безпечними.

3. Створити програму, котра буде виконувати задачі за обраним варіантом,

використовуючи написаний студентом пул потоків. Зі спільних

характеристик: код відповідальний за додавання задач в пул потоків, та сам

пул потоків повинні знаходитися в різних потоках виконання.

4. Перевірити та довести коректність роботи програми з використанням

системи вводу/виводу інформації в консоль. Зробити обмежене за часом

тестування та розрахувати кількість створених потоків та середній час

знаходження потоку в стані очікування. Визначити середню довжину

кожної черги та середній час виконання задачі (для необмежених черг).

Або визначити максимальний та мінімальний час, поки черга була

заповнена, кількість відкинутих задач (для обмежених черг).

**Варіант завдання**: 10. Пул потоків містить дві черги, перша обслуговується 3-ма потоками, а друга – 1-им. Задачі додаються в першу чергу виконання через один інтерфейс (користувач не має явного доступу до черг виконання) в порядку «складності» (коротші за часом задачі мають більший пріоритет). Якщо задача з першої черги не виконується на протязі подвійного часу, необхідного для її виконання, вона переноситься в другу чергу. Задача береться на виконання з обох черг одразу за наявності відповідного вільного робочого потоку. Задача займає випадковий час від 5 до 10 секунд.

Хід роботи

1. Пул потоків (thread pool) - це механізм управління потоками, який дозволяє ефективно використовувати та перевикористовувати обмежену кількість потоків для виконання задач у багатопотокових програмах.

У багатопотокових програмах, коли потрібно виконати багато завдань паралельно, створення нового потоку для кожного завдання може призвести до великого навантаження на систему та витрати ресурсів на створення та знищення потоків. Пул потоків дає змогу уникнути цих недоліків, створюючи пул заздалегідь ініціалізованих потоків, які готові виконувати завдання.

У пулі потоків є обмежена кількість потоків, наприклад, п'ять. Коли програма подає нову задачу для виконання, пул потоків призначає один з потоків для виконання цієї задачі. Якщо всі потоки вже зайняті, задача ставиться в чергу і очікує на вільний потік. Коли потік завершує виконання задачі, він повертається до пулу та може бути призначений для виконання наступної задачі.

Використання пулу потоків дозволяє ефективно використовувати наявні ресурси системи, зменшує накладні витрати на створення та знищення потоків, а також дозволяє керувати кількістю одночасно виконуваних потоків, що може бути корисним для уникнення перевантаження системи.

2. Реалізований пул потоків

class ThreadPool {

public:

ThreadPool(size\_t numThreads, size\_t numThreads\_1) : stop(false) {

for (size\_t i = 0; i < numThreads; ++i) {

threads.emplace\_back([this] {

while (true) {

Task task;

{

std::unique\_lock<std::mutex> lock(mutex);

condition.wait(lock, [this] { return stop || !tasks.empty(); });

if (stop && tasks.empty()) return;

task = std::move(tasks.top());

tasks.pop();

}

task.function();

}

});

}

for (size\_t i = 0; i < numThreads\_1; ++i) {

threads.emplace\_back([this] {

while (true) {

Task task;

{

std::unique\_lock<std::mutex> lock(mutex\_1);

condition\_1.wait(lock, [this] { return stop || !tasks\_1.empty(); });

if (stop && tasks\_1.empty()) return;

task = std::move(tasks\_1.front());

tasks\_1.pop();

}

task.function();

}

});

}

}

template <class F, class... Args>

void enqueue(std::chrono::seconds priority, F&& f, Args&&... args) {

{

std::unique\_lock<std::mutex> lock(mutex);

std::function<void()> function = [=]() { std::move(f)(std::move(args)...); };

transfer\_tasks();

tasks.emplace(priority, std::move(function));

}

condition.notify\_one();

}

void transfer\_tasks() {

std::priority\_queue<Task, std::vector<Task>, std::greater<Task>> new\_tasks;

while (!tasks.empty()) {

const Task& task = tasks.top();

if (task.time\_to\_postpone()) {

std::unique\_lock<std::mutex> lock(mutex\_1);

tasks\_1.push(task);

}

else {

new\_tasks.push(task);

}

tasks.pop();

}

tasks = std::move(new\_tasks);

condition\_1.notify\_one();

}

~ThreadPool() {

{

std::unique\_lock<std::mutex> lock(mutex);

stop = true;

}

condition.notify\_all();

condition\_1.notify\_all();

for (std::thread& thread : threads) {

thread.join();

}

}

private:

std::vector<std::thread> threads;

std::priority\_queue<Task, std::vector<Task>, std::greater<Task>> tasks;

std::queue<Task> tasks\_1;

std::mutex mutex;

std::mutex mutex\_1;

std::condition\_variable condition;

std::condition\_variable condition\_1;

bool stop;

};

Код представляє реалізацію класу ThreadPool, який представляє пул потоків. Давайте розберемо, як працює кожна функція:

1. Конструктор `ThreadPool`:

- Приймає два аргументи: `numThreads` та `numThreads\_1`, які вказують кількість потоків, створюваних у пулі.

- Ініціалізує змінну `stop` як `false`, яка використовується для визначення, коли потрібно зупинити виконання потоків.

- Створює потоки та додає їх у вектор `threads`. Потоки виконують нескінченний цикл, у якому вони витягують завдання з відповідної черги та виконують їх функції.

- Перший цикл створює потоки, які отримують завдання з пріоритетної черги `tasks`.

- Другий цикл створює потоки, які отримують завдання зі звичайної черги `tasks\_1`.

2. Функція `enqueue`:

- Приймає аргументи `priority`, `f` та `args`.

- Створює лямбда-функцію `function`, яка викликає передану функцію `f` з переданими аргументами `args`.

- Блокує м'ютекс `mutex`, щоб безпечно додати завдання до черги.

- Викликає функцію `transfer\_tasks()`, яка переносить завдання з основної черги в `tasks\_1`, якщо настав час виконання.

- Додає завдання із заданим пріоритетом у пріоритетну чергу `tasks`.

- Викликає `notify\_one()` для сповіщення одного потоку про наявність завдання.

3. Функція `transfer\_tasks`:

- Створює нову пріоритетну чергу `new\_tasks`.

- Поки черга `tasks` не порожня, витягує завдання з вершини та перевіряє, чи настав час виконання для завдання.

- Якщо час виконання настав, блокує м'ютекс `mutex\_1` та поміщає завдання в чергу `tasks\_1`.

- Інакше поміщає завдання в нову пріоритетну чергу `new\_tasks`.

- Замінює поточну чергу `tasks` на нову чергу `new\_tasks`.

- Викликає `notify\_one()` для повідомлення одного потоку з пулу, що з'явилося нове завдання у черзі `tasks\_1`.

4. Деструктор `~ThreadPool`:

- Блокує м'ютекс `mutex` для безпечної зміни змінної `stop` на `true`.

- Викликає `notify\_all()` для сповіщення всіх потоків про необхідність зупинки.

- Очікується завершення всіх потоків за допомогою `join()`.

Вся логіка роботи пула потоків полягає в постійній обробці завдань з черг tasks і tasks\_1 в кожному потоці. При надходженні нового завдання вона додається у відповідну чергу, і один із потоків витягує завдання та виконує її функцію.

3.

void taskFunction(int id, int time) {

std::unique\_lock<std::mutex> lock(mut);

std::cout << "Task " << id << " started with time " << time << " seconds" << std::endl;

lock.unlock();

std::this\_thread::sleep\_for(std::chrono::seconds(time));

lock.lock();

std::cout << "Task " << id << " finished" << std::endl;

lock.unlock();

}

void filling\_queue(ThreadPool\* pool, int num\_work, std::mt19937& generator) {

for (int i = 0; i < num\_work; ++i) {

std::chrono::seconds time(5 + generator()%6);

if (i > num\_work / 2)

std::this\_thread::sleep\_for(std::chrono::seconds(3+ generator()%3));

pool->enqueue(time, taskFunction, i, time.count());

}

}

int main() {

ThreadPool pool(3, 1);

std::mt19937 generator((unsigned int)std::chrono::system\_clock::now().time\_since\_epoch().count());

std::thread mythread(filling\_queue, &pool, 20, std::ref(generator));

mythread.join();

return 0;

}

1. Функція `taskFunction`:

- Приймає два аргументи `id` та `time`.

- Створює об'єкт `std::unique\_lock<std::mutex>` з м'ютексом `mut`, щоб захопити м'ютекс та забезпечити потокобезпечний вивід у консоль.

- Виводить повідомлення про початок завдання із заданим `id` та `time`.

- Розблокує м'ютекс після виведення.

- Зупиняє виконання потоку на час `time` з використанням `std::this\_thread::sleep\_for()`, щоб імітувати виконання завдання.

- Захоплює м'ютекс знову перед виведенням повідомлення про завершення завдання із заданим `id`.

- Виводить повідомлення про завершення завдання.

- Розблокує м'ютекс після виведення.

2. Функція `filling\_queue`:

- Приймає покажчик на об'єкт `ThreadPool`, кількість завдань `num\_work` та генератор випадкових чисел `generator`.

- У циклі створює завдання та додає їх у пул потоків.

- Генерує випадковий час виконання `time` в діапазоні від 5 до 10 секунд.

- Якщо номер задачі `i` більше половини загальної кількості завдань, потік, що викликає `filling\_queue`, зупиняється на випадковий час від 3 до 5 секунд за допомогою `std::this\_thread::sleep\_for()`, щоб імітувати затримку у надходженні задач.

- Викликає функцію `enqueue` об'єкта `ThreadPool` для додавання завдання із заданим часом виконання та функцією `taskFunction`.

- Після завершення циклу всі завдання додані в пул потоків.

3. Функція `main`:

- Створює об'єкт `ThreadPool` з 3 потоками та 1 додатковим потоком.

- Створює генератор випадкових чисел `generator` із використанням поточного часу.

- Створює потік `mythread`, який викликає функцію `filling\_queue` з покажчиком на об'єкт `ThreadPool`, кількістю завдань 20 та генератором випадкових чисел.

- Очікує завершення потоку `mythread` за допомогою `join()`.

- Повертає 0, завершуючи програму.

В результаті виконання програми пул потоків `ThreadPool` створюється з 3 потоками для виконання завдань з пріоритетної черги `tasks` та 1 потоком для виконання завдань із звичайної черги `tasks\_1`. Функція `filling\_queue` додає 20 завдань у пул потоків з різними часами виконання. Потім потік `mythread` завершує свою роботу, і програма завершується, чекаючи завершення всіх завдань у пулі потоків.

**Висновок**: в ході виконання лабораторної роботи було створенно пул потоків з 2 чергами для задач: першу чергу обслуговує 3 потоки, другу 1 потік. Задачі додаються в першу чергу виконання через один інтерфейс (користувач не має явного доступу до черг виконання) в порядку «складності» (коротші за часом задачі мають більший пріоритет). Якщо задача з першої черги не виконується на протязі подвійного часу, необхідного для її виконання, вона переноситься в другу чергу. Задача береться на виконання з обох черг одразу за наявності відповідного вільного робочого потоку. Задача займає випадковий час від 5 до 10 секунд. Операції ініціалізації та знищення пулу, додавання та вилучення задач в чергу є потіко-безпечними.

Лістинг

#include <iostream>

#include <vector>

#include <queue>

#include <thread>

#include <functional>

#include <mutex>

#include <condition\_variable>

#include <random>

#include <chrono>

std::mutex mut;

struct Task {

std::chrono::seconds priority;

std::function<void()> function;

std::chrono::steady\_clock::time\_point creationTime;

Task() : creationTime(std::chrono::steady\_clock::now()) {}

Task(std::chrono::seconds p, std::function<void()> f)

: priority(p), function(std::move(f)), creationTime(std::chrono::steady\_clock::now()) {}

bool time\_to\_postpone() const {

std::chrono::steady\_clock::time\_point currentTime = std::chrono::steady\_clock::now();

std::chrono::duration<double> elapsedTime = currentTime - creationTime;

if (elapsedTime >= 2 \* priority)

return true;

else

return false;

}

bool operator<(const Task& other) const {

return priority < other.priority;

}

bool operator>(const Task& other) const {

return priority > other.priority;

}

};

class ThreadPool {

public:

ThreadPool(size\_t numThreads, size\_t numThreads\_1) : stop(false) {

for (size\_t i = 0; i < numThreads; ++i) {

threads.emplace\_back([this] {

while (true) {

Task task;

{

std::unique\_lock<std::mutex> lock(mutex);

condition.wait(lock, [this] { return stop || !tasks.empty(); });

if (stop && tasks.empty()) return;

task = std::move(tasks.top());

tasks.pop();

}

task.function();

}

});

}

for (size\_t i = 0; i < numThreads\_1; ++i) {

threads.emplace\_back([this] {

while (true) {

Task task;

{

std::unique\_lock<std::mutex> lock(mutex\_1);

condition\_1.wait(lock, [this] { return stop || !tasks\_1.empty(); });

if (stop && tasks\_1.empty()) return;

task = std::move(tasks\_1.front());

tasks\_1.pop();

}

task.function();

}

});

}

}

template <class F, class... Args>

void enqueue(std::chrono::seconds priority, F&& f, Args&&... args) {

{

std::unique\_lock<std::mutex> lock(mutex);

std::function<void()> function = [=]() { std::move(f)(std::move(args)...); };

transfer\_tasks();

tasks.emplace(priority, std::move(function));

}

condition.notify\_one();

}

void transfer\_tasks() {

std::priority\_queue<Task, std::vector<Task>, std::greater<Task>> new\_tasks;

while (!tasks.empty()) {

const Task& task = tasks.top();

if (task.time\_to\_postpone()) {

std::unique\_lock<std::mutex> lock(mutex\_1);

tasks\_1.push(task);

}

else {

new\_tasks.push(task);

}

tasks.pop();

}

tasks = std::move(new\_tasks);

condition\_1.notify\_one();

}

~ThreadPool() {

{

std::unique\_lock<std::mutex> lock(mutex);

stop = true;

}

condition.notify\_all();

condition\_1.notify\_all();

for (std::thread& thread : threads) {

thread.join();

}

}

private:

std::vector<std::thread> threads;

std::priority\_queue<Task, std::vector<Task>, std::greater<Task>> tasks;

std::queue<Task> tasks\_1;

std::mutex mutex;

std::mutex mutex\_1;

std::condition\_variable condition;

std::condition\_variable condition\_1;

bool stop;

};

void taskFunction(int id, int time) {

std::unique\_lock<std::mutex> lock(mut);

std::cout << "Task " << id << " started with time " << time << " seconds" << std::endl;

lock.unlock();

std::this\_thread::sleep\_for(std::chrono::seconds(time));

lock.lock();

std::cout << "Task " << id << " finished" << std::endl;

lock.unlock();

}

void filling\_queue(ThreadPool\* pool, int num\_work, std::mt19937& generator) {

for (int i = 0; i < num\_work; ++i) {

std::chrono::seconds time(5 + generator()%6);

if (i > num\_work / 2)

std::this\_thread::sleep\_for(std::chrono::seconds(3+ generator()%3));

pool->enqueue(time, taskFunction, i, time.count());

}

}

int main() {

ThreadPool pool(3, 1);

std::mt19937 generator((unsigned int)std::chrono::system\_clock::now().time\_since\_epoch().count());

std::thread mythread(filling\_queue, &pool, 20, std::ref(generator));

mythread.join();

return 0;

}